The Remote Agent Executive: Capabilities to Support Integrated Robotic Agents

Barney Pell[§] Gregory A. Dorais[†] Christian Plaunt[†] Richard Washington[†]
{pell,gadorais,plaunt,richw}@ptolemy.arc.nasa.gov
NASA Ames Research Center, Mail Stop 269-2
Moffett Field, CA 94035-1000

July 6, 1998

Abstract

The Remote Agent (RA) integrates a broad spectrum of robotic activities, including planning, scheduling, execution, monitoring, failure detection, diagnosis, and recovery. The RA Executive (EXEC) can be viewed as the core of the agent. EXEC enables software developers to think about the robot at a higher level; it also supports the reuse of knowledge and code across multiple robot applications. EXEC's capabilities include a high-level procedural action-definition language, services for resource management and configuration management, support for executing flexible closed-loop plans and command sequences, support for replanning, and a framework for specifying fault responses including safe modes and responses to plan failures.

We believe that these capabilities are required for most autonomous agents, and that executives and execution capabilities will become more essential as we attempt to develop autonomous agents of increasing capability. Moreover, we deem modularity, variable autonomy, robustness, and support for tools and testing to be general architectural properties necessary for design, deployment, and operation of executives within a complex mission. To this end, we are remodularizing the executive by providing each capability as a separate component so that the individual capabilities can be standardized and shared across different agent architectures.

1 Introduction

An executive coordinates the runtime activity among a set of modules involved in a control system. Such coordination is necessary to synchronize coordinated activities, avoid resource conflicts, enforce operational constraints, and ensure safety in the presence of failures.

The capability level of an executive has direct impact on the design and complexity of the entire system. If there is no executive in the system, the individual modules must be commanded separately. Also, all the coordination requirements must be performed open-loop or the modules must know about each other and coordinate themselves directly. This increases the complexity of both the external commanding system and the functional modules themselves.

Most deployed spacecraft control systems today have a simple sequencing engine on-board, which accepts a timed sequence of commands and issues those commands to the functional modules. The current systems do not support context-sensitive or conditional activity, or decomposition of high-level activities into lower-level commands. One result is that the duration of activities must be determined open-loop by the ground system. These limitations cause significant complexity in the ground system, as it must reason about every

[§]Research Institute for Advanced Computer Science
[†]Caelum Research Corporation

detail and every interaction of the control system. Even for ground systems that employ automated planners to ease the burden, the level of detail required for planning down to the lowest-level activities makes the planning problem computationally difficult. The low-level detail also makes the knowledge encoded and used by the ground system (including the planner) cumbersome to extract and validate, and error prone.

Moreover, since current executives do not understand the causal structure of the sequences they execute and do not perform context-sensitive activities, the resulting behavior is brittle. The sequence tends to break if anything goes wrong, causing loss of mission goals and missed opportunities. Missions that cannot afford such losses at critical periods wind up pushing further complexities into the functional module software, and onto the fault-protection software. For example, in the "sequence rollback" approach, the functional modules that receive the low-level commands must be able to execute the commands differently depending on whether they succeeded or failed last time, so that the same sequence can be re-used in the case of failures. This approach leads to software that is almost incomprehensible, unmaintainable, and certainly not reusable.

In this paper, we describe execution capabilities and architectural properties that are important for developing, deploying, and operating autonomous spacecraft. We believe that these capabilities are required for most autonomous agents, and that executives and execution capabilities will become more essential as we attempt to develop autonomous agents of increasing capability.

1.1 The Remote Agent

To address the limitations of spacecraft control systems, we have developed the Remote Agent (RA). RA is designed to control spacecraft autonomously for extended periods of time, and is being demonstrated on the Deep Space One spacecraft (DS-1) scheduled to launch October, 1998 [Bernard *et al.*, 1998]. A prototype of RA was also tested on a complex simulated Saturn Orbit Insertion (SOI) scenario for the Cassini spacecraft [Pell *et al.*, 1998a]. However, RA is not limited to controlling spacecraft. RA is a general agent architecture that is applicable to a wide variety of applications; we are currently applying it to communication satellites and rovers, with possible future applications to life support systems, autonomous propellant manufacturing, and space and earth based interferometers.

The Remote Agent itself comprises four components: a Mission Manager (MM), a Planner/Scheduler (PS), a Smart Executive (EXEC), and a Mode Identification and Reconfiguration component (MIR) [Pell *et al.*, 1998a, Muscettola *et al.*, 1998b]. The architecture is shown in Figure 1. This paper focuses on EXEC.

Mission Manager (MM): MM is special module of the planner that formulates shorter-term planning problems based on a long-range *mission profile*. The mission profile, which is provided at launch, contains a list of all nominal goals to be achieved during the mission. MM determines the goals that need to be achieved in the next scheduling horizon (typically 2 weeks long), extracts them from the mission profile and combines them with the initial (or projected) spacecraft state as determined by the executive. The result is a specific planning problem that, once solved, yields detailed execution commands. This decomposition into long-range mission planning and shorter-term detailed planning enables RA to undertake an extended diverse mission with minimal human intervention.

Planner/Scheduler (PS): PS is an integrated planner and scheduler. In our architecture, PS is activated as a "batch process" that terminates after a new schedule has been generated. It takes as input a *plan-request* and produces as output a flexible, concurrent temporal plan. A plan-request describes the current state of execution, including activities still scheduled for the future, as well the goals for the next plan execution period as formulated by MM.

PS constructs plans using a knowledge base expressing domain constraints and heuristics. Other on-board software systems, called *planning experts*, participate in the planning process by requesting new goals or answering questions asked by the PS. For example in the DS-1 RA, the navigation system planning expert requests a set of pictures it will need to update its understanding of the current spacecraft trajectory, and the attitude planning expert answers questions about estimated duration of turns and resulting resource consumption, given assumptions expressed by the planner about a hypothetical planning context.

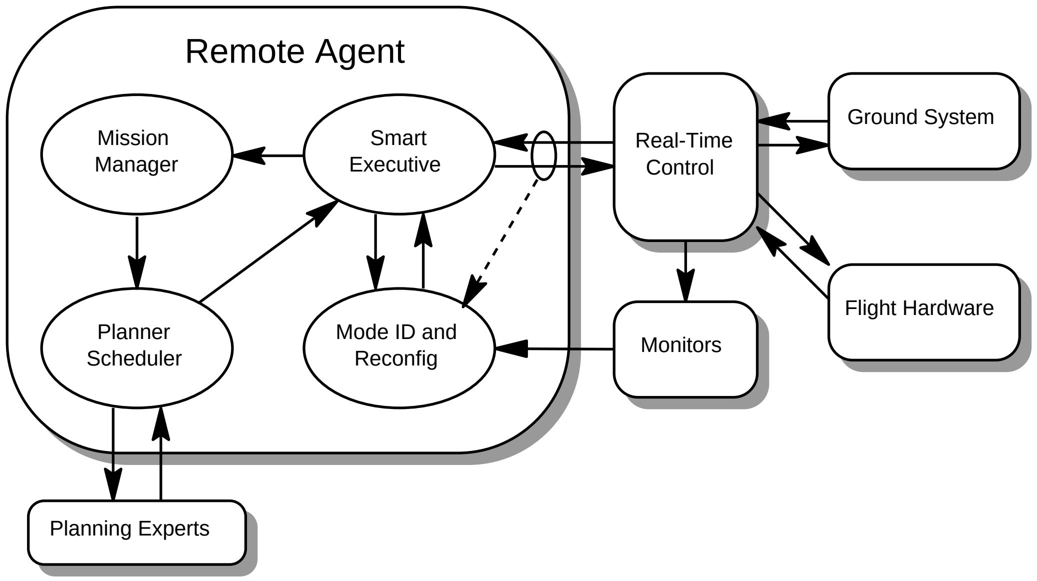

Figure 1: Remote Agent architecture

An output plan constrains the activity of each spacecraft subsystem over the duration of the plan, but leaves flexibility for details to be resolved during execution. The plan also contains activities and information required to monitor the progress of the plan as it is executed. In addition, the plan contains an explicit activity to initiate another round of planning.

Smart Executive (EXEC): EXEC is a reactive, plan-execution system with responsibilities for coordinating execution-time activities, including resource management, action definition, fault recovery, and configuration management. These will be discussed in more detail below.

Mode Identification and Reconfiguration (MIR): MIR is a discrete, model-based controller that uses a single, declarative model of the spacecraft for both mode identification and reconfiguration. MIR is based on Livingstone as described in [Williams and Nayak, 1996]. Like EXEC, MIR runs as a concurrent reactive process. MIR itself contains two components, one for *Mode Identification* (MI) and one for *Mode Reconfiguration* (MR). MI is responsible for providing a level of abstraction to the executive that enables EXEC to reason about spacecraft state in terms of a set of component modes rather than a set of low-level sensor readings. MR serves as a *recovery expert* to EXEC, taking as input a *recovery request*, and returning a sequence of operations that, when executed starting in the current state, will move the executive into a state satisfying the properties required for successful execution of the failed activity.

1.2 The Remote Agent Executive

EXEC is event-driven and goal-oriented and forms the core of RA. It provides a language and a framework in which software designers can express how planning, control, diagnosis, and reconfiguration capabilities are to be integrated into an autonomous system. It can request and execute plans involving concurrent activities that may be interdependent, where the success, timing, and outcomes of these activities may be uncertain. It provides a language for expressing goal-decompositions and resource interactions. When interpreting this language at run time, the executive automates the decomposition of goals into smaller activities that can be executed concurrently. This automates aspects of the labor-intensive sequencing function in spacecraft operations and raises the level of abstraction at which the ground system or on-board planner must reason.

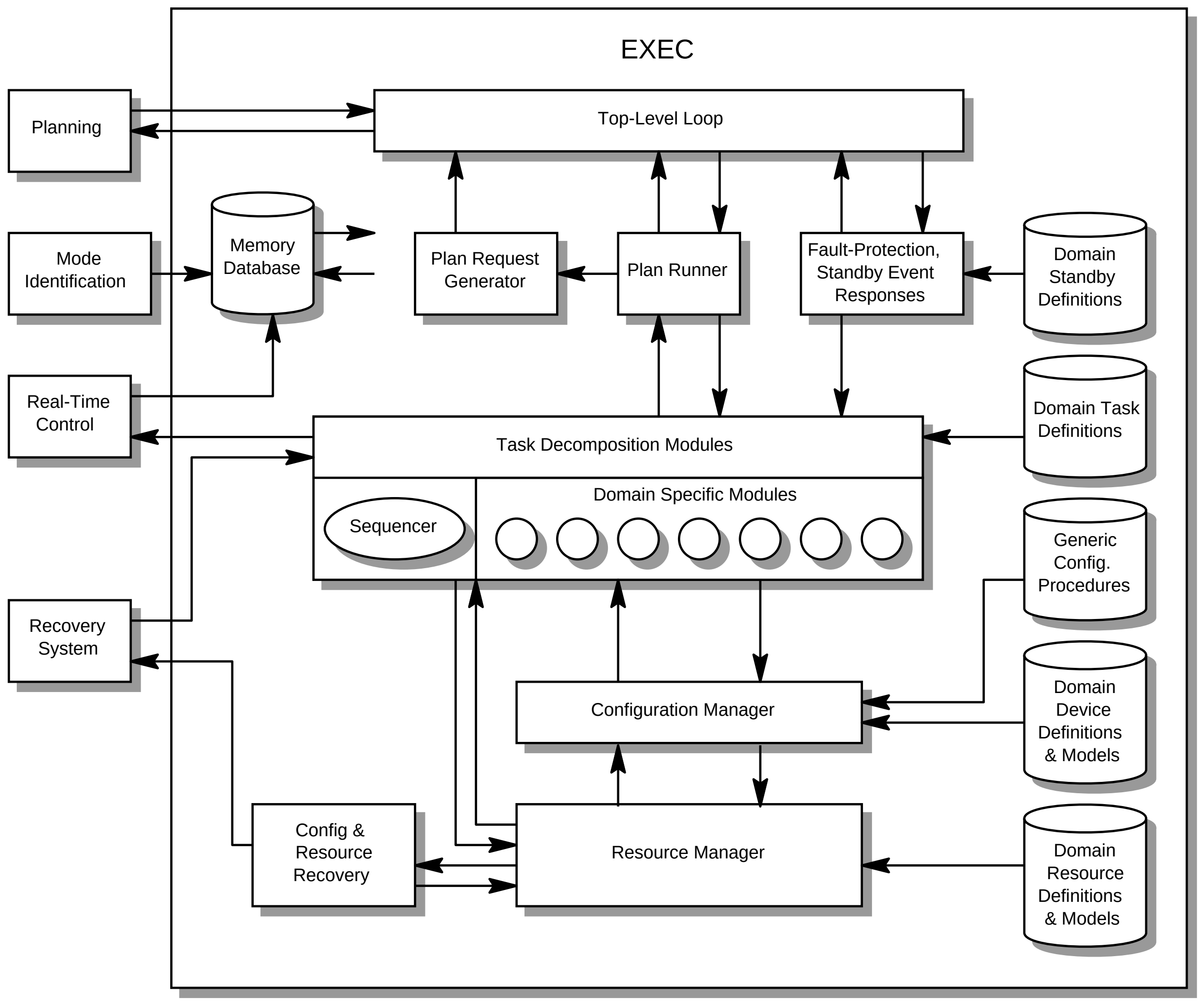

Figure 2: The architecture of the Smart Executive (EXEC)

The framework also supports a close integration between activity decomposition and fault responses. This leads to more robust execution, avoids loss of mission objectives, improves mission reliability and resource utilization, and simplifies the design of the entire software system.

The architecture of EXEC is illustrated in figure 2 and is an expansion of the Smart Executive node shown in figure 1. Note that the domain definitions and models are separate from the EXEC components allowing users of EXEC to tailor it for a variety of domains without having to modify any of the EXEC components. Also note that each EXEC component can read and write to the memory database as required. The main EXEC components are discussed in detail in the next section.

As an integrating technology, an executive enables each software component to be expressed independently, but controls the interactions among those components so that the overall activity of the system achieves global constraints as dictated by a high-level plan. This increase in modular software design enables reduced development and integration costs and increased software reuse both within and across missions. This capability also enables a new generation of adaptive controllers with multiple control modes. A smart executive switches control modes of these controllers based on feedback from the environment.

Our understanding of the role and requirements of executive capabilities has evolved since we began our prototyping effort on the SOI scenario. That work built on executive components the Cassini engineers had developed, including a configuration manager, resource manager, and fault-protection framework [Brown *et al.*, 1995] and extended these capabilities to support an on-board planner and model-based diagnosis and repair system. On DS-1, we moved from a prototype to an operational flight software system that was part of a much larger project. This experience provided lessons about additional requirements that were critical to successful deployment in an actual mission [Aljabri *et al.*, 1998]. Our current work revolves around extending the capabilities of the individual components and addressing critical features in response to the lessons from our flight experience.

The rest of this paper is structured as follows. The next section describes EXEC components and capabilities we think are general to most autonomous control systems. This is followed by a section where we present general architectural properties necessary for design, deployment, and operation within a complex mission. We then discuss related work and future directions.

2 EXEC Components

EXEC has the following capabilities which we consider useful for executives in general:

- Flexible plan execution
- Configuration management
- Resource management
- Action-definition language
- System-level fault-protection support

While these capabilities are tightly coupled within EXEC, we will describe each as a separate component with independent functionality. We are currently redesigning EXEC to reflect this modular architecture, as discussed later. RA components that implement these capabilities are illustrated in Figure 2.

2.1 Flexible Plan Execution

One major capability of EXEC is to support execution of plans generated by PS. Several features of plan execution are important in our applications.

An important set of features concerns the plan representation itself. PS generates *flexible, concurrent, temporal plans* [Pell *et al.*, 1998a, Muscettola *et al.*, 1995]. This is important for control systems in which many subsystems conduct concurrent activity, and in which execution must be robust in the face of uncertainty. Despite the flexibility in the plans, it is important that EXEC execute the plans correctly and efficiently. EXEC's own processing of the plan introduces some latency during execution. This introduces issues of temporal granularity and propagation times into the system design [Muscettola *et al.*, 1998a].

An illustration of a simple plan is shown in figure 3. Each horizontal bar represents a separate thread of activity. Each thread is segmented into a sequence of activities that are executed from left to right. The plan contains constraints that coordinate the execution of activities on different threads. For example, the imaging activity on the Imaging thread shown in the figure has a *contained-by* constraint with reference to the point(b) activity on the Attitude thread. This means that the imaging activity cannot start until the point(b) activity has started and the imaging activity must end before the point(b) activity ends.

Planning can be an infrequent activity due to resource and informational limitations. Hence, EXEC supports the execution of infrequently-generated plans [Pell *et al.*, 1997]. To avoid interrupting activities at the boundary between execution of one plan and the next, EXEC supports *smooth plan transitions*, in which the same activities can be referred to between subsequent plans. Thus, a new plan can be smoothly merged into the previous execution context.

EXEC supports *concurrent planning and execution*, in which a new plan can be generated while a previous plan is executing [Pell *et al.*, 1997]. The planning process itself is represented as part of the plan being

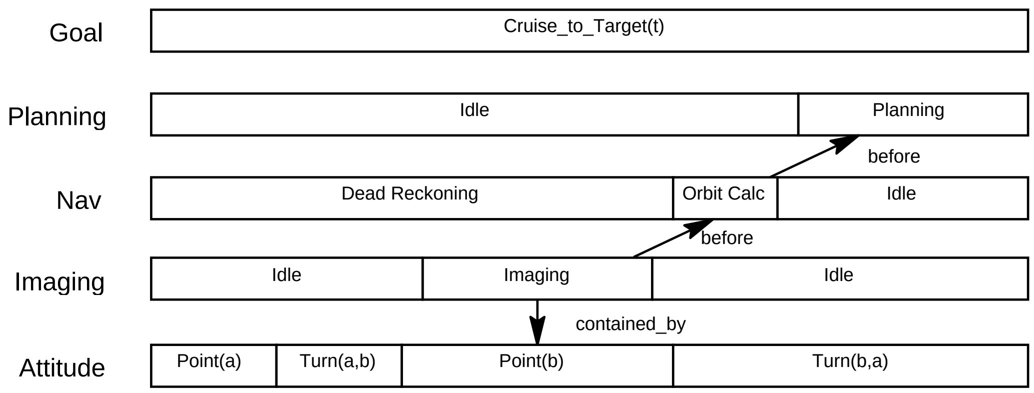

Figure 3: Planning as a constrained execution activity.

executed, and thus the new plan will be generated at a time constrained by other events in the plan. For example, in the plan shown in figure 3, the planning activity cannot start until after the orbit calculation activity has completed.

Planning during execution requires the planner to make use of current execution information. Also, EXEC must track the assumptions under which the new plan is being generated, and abort the planning process if those assumptions become invalid before the new plan is executed.

Monitoring the execution and aborting (under conditions of failure or by command) is also an important requirement. When the execution of a plan is aborted, EXEC terminates activities executing under the current plan (including any replanning activity currently underway) and invokes a procedure to establish a stable configuration, called a *standby mode*, from which a new round of planning can be initiated [Pell *et al.*, 1997].

EXEC supports a mechanism by which users can switch between closed-loop plan execution, in which EXEC requests plans based on current state information, and open-loop execution, in which EXEC runs plans possibly generated by mission operators and sent to the spacecraft [Pell *et al.*, 1998c]. For this feature to be useful, it is important to support smooth plan transitions in the open-loop context as well.

In addition to the ability to execute high-level plans, it is also important to support the functionality of traditional sequencers. While such sequences are less flexible (and thus less robust in the face of uncertainty), they can be easier to analyze and predict, and hence more comfortable for mission operators. EXEC supports the ability to run traditional sequences, and also to embed sequences within flexible plans. This enables a form of adjustable autonomy (discussed later) in which operators can choose to control some aspects of operation via high-level goals managed by the agent but control other aspects by low-level commanding. Even when EXEC is running sequences, it gives operators access to a rich language in which they can express time-driven, event-driven, and context-sensitive relationships between activities.

2.2 Services for Configuration Management

A challenge in designing software for a complex system, such as a spacecraft, is that the system may be in a wide variety of configurations. The physical components of a spacecraft, their topology, and the state of each component define a spacecraft configuration. At hardware design time, devices may be added or removed from the system (for example, a new star tracker), and details of individual hardware components may change. At run time, devices may be turned on or turned off, healthy or malfunctioning, and in a number of possible operational states.

EXEC's configuration manager (CM) insulates the software for each of the many activities in a complex system from the details of the configurations that those activities require. If the configuration details change, whether at run time or design time, the activity code can remain unchanged.

CM maps configuration requirements to code that establishes configurations satisfying those requirements. The code that establishes configurations must reason about inter-dependencies among devices, and it must adapt to hardware changes. To simplify development of such code, we have designed a set of generic configuration management procedures, in which the user provides hardware details (including connectivity diagrams) and instantiates the generic procedures.

Once a configuration satisfying the required properties is established, CM tracks this configuration and signals an error if these properties are violated. In such failure conditions, CM interacts with the fault-protection capabilities to establish a potentially new configuration so that the code requiring it can continue to operate correctly.

In EXEC, individual device knowledge is implemented based on a library of generic device management routines. Devices and classes are formalized using generic descriptions. Individual devices, switches, etc., are then modeled as instances of these classes. Consider the following example:

```
(define-device-class :camera
  :power-function #'fsc-power-request
  :talk-function #'camera-talk-msg)

(define-device :camera_A :camera
 :powered-thru :power_bus_1
 :switched-thru :fsc_camera_sw1
 :ready-state ((:health_state :ok)
               (:power_state :on)))
```

Based on these device idioms, EXEC has generic procedures defined for device configuration and management. With a single construct, CM will select a device of a given class, achieve its ready-state, and then lock the properties of the ready-state and maintain them during the execution of an activity. Note that achieving a state of one device may require readying other devices it depends on, recursively.

Based on the camera definition above, the construct

```
(with-selected-device :camera
  (take-pictures))
```

would select a camera (say *camera_A*), achieve its ready-state of being powered on and healthy, and then take pictures within a context that ensures that the health and power of the camera are maintained throughout picture taking.

2.3 Services for Resource Management

Scarce resources are a fact of life on spacecraft. On-board processes contend for a limited number of devices. Shared resources, such as energy and data storage, are subject to hard (and relatively severe) limits, as well as environmental influences (such as solar exposure for battery charging) that change these limits over time.

Currently, spacecraft control sequences use worst-case estimates of resource utilization. However, the uncertainty in the environment and its effect on the true resource utilization can make estimates inaccurate. For example, in the rover domain, navigation over a fixed distance can use widely varying amounts of energy depending on the terrain the temperature. A loose worst-case estimate can lead to under-utilization of the spacecraft; an underestimate can lead to aborts in plan execution.

A spacecraft resource manager must be able to manage and coordinate utilization of these scare resources. First, it must provide a basic locking mechanism that maintains devices in particular states throughout the time interval for which the device state is needed by a process. Second, it must handle continuous and varying resources by responding to immediate resource requests from tasks, tracking resources over time, and allowing resources to be reserved for future tasks.

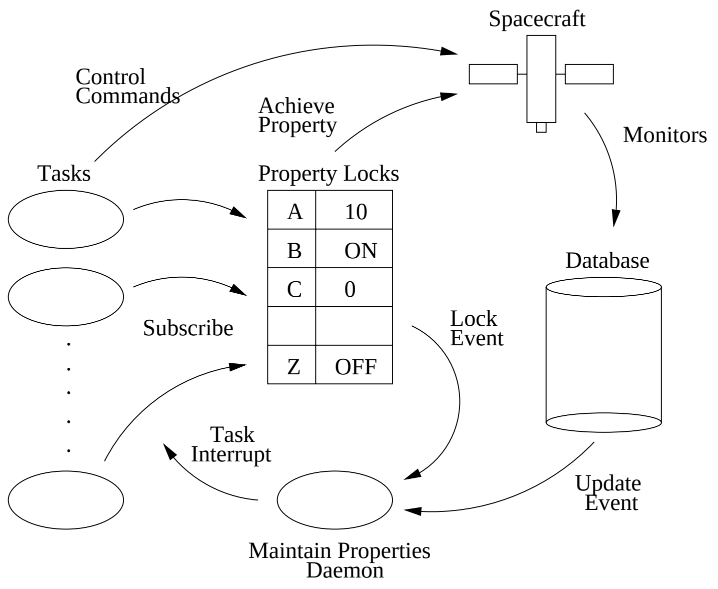

Figure 4: Procedural Executive Resource Manager

2.3.1 Property locks

The executive manages a set of concurrent control tasks, as shown in Figure 4. Each control task requires a set of *resources*, or *properties*, to be established and maintained over some period of time. For example, the activity of taking pictures with a camera requires that the camera is on and functional. If some other activity requires the camera to be off, these two activities *compete* for the resource of controlling the camera's power state. The executive must achieve, maintain, and monitor properties required for each task, and resolve task resource conflicts.

A task is represented at run-time by an independent execution *thread*. Threads communicate with other threads directly via *signals*, or indirectly via changes to a *database*. Receipt of a signal or notification of a change to the database are examples of *events*.

Each activity uses the `(with-maintained-properties)` construct to declare those properties that it requires maintained over its interval of execution. In this way, Exec understands the constraints which support the entire current execution context. When a property is achieved and reserved for a task, it is said to be *locked* until the task relinquishes it, so that other tasks will not be permitted to violate that property. Of course, the locks reflect properties true in the current state, and sometimes these properties can change despite the best efforts of the software system to maintain them. For example, switches on a spacecraft sometimes change state accidentally. In this case, we describe the properties as *lost* or *violated*, and the tasks requiring them as *unsupported*.[1]

In the event that some property is lost or otherwise unachievable without the help of a recovery expert, Exec suspends the unsupported threads, formulates a query based on the active constraints, and uses the `automatic-recoveries` thread to send the query off to the recovery expert (in this case, MIR).

When the recovery expert returns an action, Exec performs the action and then re-activates any suspended threads which may now be supported. The threads then attempt to re-establish their maintained conditions. Note that most Exec procedures count the number of times they have retried a particular approach, and try something else or give up if this retry counter exceeds a threshold.

[1]Note that property locks can serve a role similar to typical locks in multi-threaded systems, such as semaphores and mutexes. However, there is a major difference since these property locks are database-relative, and can hence be "taken" by the outside world changing. Note also that naive use of property locks can result in deadlock, just as occurs with standard locks in multi-threaded operating systems.

The `automatic-recoveries` thread remains in action forever, so unsatisfied constraints following execution of some recovery step will lead to a new recovery request.

We now elaborate on some of the key constructs we have developed within the procedural executive that support the behavior described above.

Achieving properties

`(achieve <property>)`

- If this is the first thread to request the property, then execute an achievement method for the property.
- When achievement is successful, signal other waiting threads.
- If some other thread is already achieving the property, then wait for it to finish.
- If the property is inconsistent with a current lock, either wait for lock to be released or fail immediately (based on preferences set by the invoking thread).

Maintained Properties

`(with-maintained-properties <properties>` *body*`)`

- If *properties* are all currently true, *body* is executed.
- If *properties* are false, the executive tries to achieve them first.
- Once they are true, the executive locks the properties and executes *body*.
- If the properties become false during execution of *body*, signal this loss and let the enclosing context of *body* choose the response.

2.3.2 Aggregate Resources

The EXEC resource manager is being extended to manage more complex resource requirements. The resources are not all-or-none, as in the current resource manager, but allow partial resource allocation (as with energy or data storage). In addition, the resources may be reserved for future use, so that future tasks may be scheduled.

An example of aggregate resources can be seen in Figure 5. The resource availability is represented as a curve, representing for example solar flux over the day. Individual tasks will require resources over intervals of time, shown by the blocks under the curve in the figure. When the resource usage exceeds the resource limits, either now or at some point in the future, the resource manager needs to notify the tasks that there is a conflict and that some amount of resource needs to be freed.

The primary functions of the resource manager being developed are to detect changes in resource availability, adapt the resource allocation to respond to the resource changes, and manage resource requests and queries from tasks. The resource availability may change for a variety of reasons: degraded components (battery use), environmentally affected performance (solar cells), or other causes. The resource manager will have multiple strategies for adapting to resource changes and conflicts. These strategies will be selectable remotely and include task abortion (the simplest and most severe), shedding of low-priority tasks, and a novel idea of temporarily borrowing resources from other tasks. Examples of applications of aggregate resource management tasks are telecommunications quality of service guarantees and rover power and data storage.

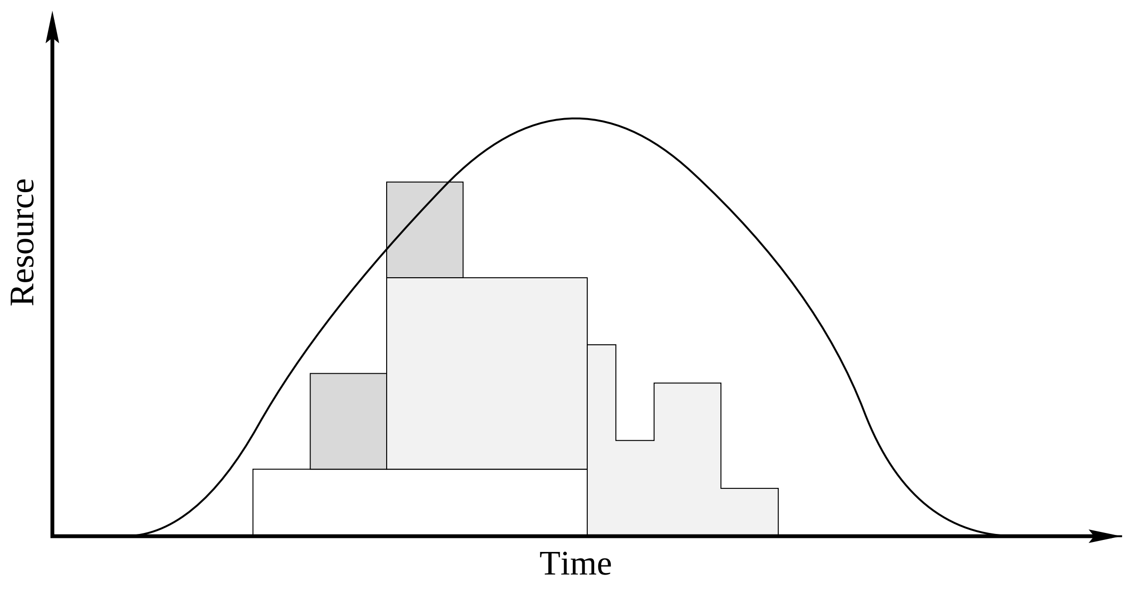

Figure 5: Aggregate resource availability and usage.

2.4 High-Level Procedural Action-Definition Language

Spacecraft control software should provide a concise way of specifying the control and behavior of the various spacecraft devices and the core components of EXEC itself, and insulate the system developer from low-level interactions with hardware and software.

EXEC is built upon a powerful, extensible, high-level language, called Execution Support Language (ESL) [Gat, 1996]. This language was developed in response to the needs of developing EXEC and the systems it controls. ESL includes the following features:

- A top-level interpreter (shell)
- Constructs for managing task networks and concurrent threads of execution
- Conditional task execution
- Robust exception handling with constructs for condition monitoring and maintenance, multiple recoveries, and cleanups
- Event signaling, response, and call-backs
- A deductive memory database
- Constructs for relative and absolute timing

A language feature we found important is *language extension*. This enables us to define new constructs which encapsulate procedures we used repeatedly and to enforce programming conventions and architectural constraints. See the camera device definition above for an example of such a language extension. The lack of this feature was one of several reasons that motivated us to develop ESL as a successor to RAPS [Firby, 1978], which we used during our prototype effort [Pell *et al.*, 1998a].

2.5 System-Level Fault Protection

Fault protection enables a system to recover from anomalous conditions such as obstacles to navigation or device malfunctions. Spacecraft engineers distinguish *component-level* fault protection, which tries to restore functionality of an individual subsystem, from *system-level* fault protection. In the latter, the overall system must be protected even if the component that failed is not identified, isolated, or recovered.

Writing fault-protection software is a significant part of a spacecraft software engineering effort because there are many subsystems and interactions to consider. To handle fault protection across subsystems, many constructs are needed. These include events and responses, timers, the ability to achieve standby mode, property tracking, constrained recoveries, and waiting for confirmation before taking action.

Our executive fault-protection component provides a robust layer for fault monitoring, detection and recovery. EXEC's fault-protection system provides:

- An event-response architecture, which has multiple threads of control listening for events (e.g., under-voltage trips or extended loss of communication), which execute well-defined responses specified in a rich language with hierarchical exception handling, context-dependent methods, etc.
- Built-in constructs that support reflex reactions to fault conditions
- An ability to draw on external expertise for monitoring, diagnosis, and recovery to handle more complex conditions and recovery plans [Pell *et al.*, 1998b]
- A high-level control loop that can abort a plan and re-plan, or robustly achieve a context-appropriate standby mode
- A fail-operational capability to restore activities that fail without breaking plans, when possible. This uses recovery methods, and can draw on the property-lock manager to help formulate constraints for use by external recovery planners
- A reusable framework for structuring system-level fault protection in the spacecraft domain, with pre-defined threads for the major subsystems (power, thermal control, communications, attitude control, etc.)

EXEC performs similar functions to a traditional operating system. The main difference is that when unexpected contingencies occur, a traditional operating system can only issue a report and abort the offending process, relying on user intervention to recover from the problem. EXEC must be able to take corrective action automatically, for example in order to meet a tight orbital insertion window.

In conjunction with an external model-based fault-diagnosis and recovery system, EXEC provides procedural constructs to control the deductive component in a "hybrid precedural/decuctive" manner [Pell *et al.*, 1998b]. The integrated capability enables designers to code knowledge using a combination of procedures and declarative models suitable to the challenges of complex systems control.

3 General Architectural Properties

This section discusses the general architectural properties necessary for design, deployment, and operation of an executive within a complex mission. Important properties include:

- Modular architecture
- Adjustable autonomy
- Robustness
- Tools for design, analysis, visualization, communication and logging
- Component testing

3.1 Modular Architecture

One major change underway to EXEC is to modularize it by separating the individual functions, while maintaining and enhancing its functionality. Advantages of a modular architecture include the following:

- The roles and relationships of the components are more cleanly defined. This makes it easier to understand the behavior and interface of each executive capability, and at the implementation level discourages subtle interactions that could otherwise occur.
- The individual components may be chosen in a plug-and-play manner to build up a customized architecture for a given application domain. This reduces the overhead when an application needs only one component, e.g., the resource manager. Plug-and-play also creates an open architecture so that other developers can produce systems satisfying the same capabilities as EXEC components.
- A system that uses executive capabilities no longer needs to be within the framework and language of the executive. When each capability is separated and has a clean interface, a system designer can use the capability with calls, written in any language, to the executive through the interface.
- Once each component capability is better understood, the opportunities for extending and improving the component are more clearly evident. Thus, the overall architecture can be upgraded in a more coherent and more useful fashion.
- Tools for the executive can be specialized to the individual components. This will enhance the effectiveness of each component, as it will allow better formal analysis, visualization, testing, and debugging of each component capability.

Our new modular executive under development, called the Intelligent Deployable Executive Architecture (IDEA), will overcome a major barrier to adoption of executive technologies within conservative space missions.

3.2 Adjustable Autonomy

An important executive property is to be able to share control of complex systems with people who interact with them. The goal of adjustable autonomy is to minimize the necessity but maximize the capability of human interaction with the system being controlled.

We distinguish two types of adjustable autonomy. *Phase-variable* autonomy permits the level of autonomy to vary over time. *Activity-variable* autonomy permits different levels of autonomy for different activities. For example, a user may which to perform one activity by tele-operation while the system performs other activities autonomously. In addition, changing the level of autonomy of a robotic agent (either phase-variable or activity-variable) may be initiated by a user or by the agent itself. For example, the executive may recognize when the system is in a situation which would benefit from human assistance and adjust itself to the appropriate level of autonomy. These forms of adjustable autonomy are discussed further in [Pell *et al.*, 1998c].

For spacecraft, adjustable autonomy increases the confidence of managers and operators responsible for multi-million dollar missions to use an autonomous executive. It also enables operators to deliver new autonomous capability incrementally over the course of a mission. Moreover, operators may desire fully autonomous operation during some phases (like cruising to a target), but little or no autonomous operation during others (like encountering a target). The ability to draw on human expertise, especially in anomalous conditions, can also simplify the design of the system and increase the chance of mission success.

The need to support adjustable autonomy in addition to high levels of autonomy raises many additional design challenges. Major sources of complexity are that the human operators and the robotic agent have difficulty predicting each other's state and intentions, and their activities may interfere. This is a major area of ongoing research [Pell *et al.*, 1998c, Bonasso *et al.*, 1997b].

3.3 Robustness

In addition to supporting system-level fault protection, an executive must be internally robust in the face of hardware and software faults. Each component must be sensitive to the limits of its own domain of expertise and respond appropriately when these limits are reached. An example is that EXEC tracks the number of times plan execution has failed, and when this count exceeds a threshold EXEC ceases to request new plans and requests help from ground. Similarly, EXEC checks that the current state of the spacecraft is within the competence of the planner before requesting a plan.

3.4 Tool Support

For each EXEC component, tools are needed for design, analysis, visualization, communication, and logging. These tools are in various stages of development for different EXEC components. We refer to some of them here.

Simmons and Whelan (1997) has developed a plan execution visualizer (ExecView) compatible with our plan execution system. ExecView enables users to view the plan as it was executed, scroll through the history, and inspect individual activities to get explanations of constraints on these activities. An interesting feature is that the system supports explanations of why an activity was unable to start before a certain time. A visualization tool for resource management is currently under development.

Researchers in the Formal Methods group at NASA Ames have worked with us to formally validate some of EXEC's capabilities. Formal analysis of models of the abstract resource management component of EXEC [Gat and Pell, 1998] has established the correctness of some aspects of the system, and also located implementation problems earlier in the process [Havelund *et al.*, 1997, Penix *et al.*, 1997, Lowry *et al.*, 1997]. We are planning to extend this work to validate the new resource management component and the plan-runner.

An interesting issue in monitoring and logging behavior of execution systems is that observation overhead can alter the behavior of such time-sensitive systems. We have developed a method for monitoring and logging EXEC *without* significant impact on the runtime behavior of EXEC. The monitoring task is run separately at a relatively low priority. When a high priority task generates logging data, that information is passed in raw form to a "logging" task that will only process the data when time and resources allow. An additional complexity arises in missions with limited communication bandwidth. Such systems often need to support dynamic prioritization of monitoring data.

3.5 Component Testing

Testing an autonomous system is a major challenge because of the wide range of activity it can demonstrate. A problem with systems that carry out behavior over extended time (such as a plan-execution system running a month-long plan) is that testing in real time can be prohibitively expensive. EXEC can scale its clock to run at some fraction of real time and suspend time and resume it at a later date. An enhanced capability, called *warping*, enables EXEC to recognize idle periods and jump directly to the time when a next planned event is supposed to occur.

Formal methods have also been applied to some of EXEC's components in oder to verify their real-time behavior and execution semantics. This has helped reduce some of the enormous effort associated with traditional system validation.

Simulation is an important component of EXEC testing. To this end, models can be used which simulate the behavior of the system which EXEC is controlling at an appropriate level of abstractions. Typically, such simulations are encapsulated in a variety of scenarios which test various system and component level EXEC capabilities and direct the focus of validation.

4 Related Work

In this section we discuss work on other execution systems and components.

EXEC is currently implemented on top of ESL [Gat, 1996], and a previous version was implemented in RAPS [Firby, 1978]. These systems both provide languages to support multi-threaded task decomposition, memory databases, and event-waiting and response mechanisms. RAPS provides a tighter level of structuring, which simplifies tracing and potentially generation of action networks from planners. However, ESL provides better support for structured objects and language extension, both of which were crucial for our large software engineering project. PRS [Georgeff and Lansky, 1987] and RPL [McDermott, 1991] are similar languages that support general execution and agent construction. Interrap [Muller and Pischel, 1994] and Golog [Levesque *et al.*, 1997] provide execution languages based on logic programming.

Unlike most other general execution systems, TCA [Simmons, 1990] explicitly supports an *executive services* perspective. User-level procedures, which can be written in different languages and run on distributed platforms, can access centralized services (such as synchronization, simple resource management, and task decomposition) through well-defined interfaces. While our previous approach was to build integrated execution systems which would explicitly coordinate external software components, we are now moving EXEC to a services perspective as well. EXEC supports a higher-level set of individual capabilities than TCA, Moreover, we are taking the services perspective one step further by providing the ability for each service to be used in a stand-alone fashion and customized according to the needs of an application.

A number of systems have been developed that support specific executive capabilities described in this paper. The attitude and articulation control subsystem (AACS) on the Cassini spacecraft [Brown *et al.*, 1995, Hackney *et al.*, 1993] has explicit software modules for context-dependent command decomposition, resource management, configuration management, and fault protection. While these components are restricted to the particular needs of the mission and the system has no support for closed-loop execution of flexible plans, these Cassini executive capabilities served as a baseline for those incorporated into EXEC.

CIRCA [Musliner *et al.*, 1993] supports fault protection in the context of guaranteeing mission-critical real-time behavior. CIRCA considers a set of states, actions, and critical failures to be avoided and constructs a program consisting of a set of sense-act transitions which is then executed by a real-time controller. The execution is guaranteed to avoid failure states.

The CONFIG system [Malin, 1997, Malin and Leifker, 1991], developed at NASA Johnson Space Center, specifically supports goal-oriented activity definitions and configuration management via declarative models. The Livingstone system [Williams and Nayak, 1996] is also designed to function as a model-based manager for discrete hardware configurations. Livingstone's modeling language is based on concurrent transition diagrams. These provide an elegant representation of hardware devices, though the current system does not treat time-delays within repair plans or support procedural event response definitions, as does EXEC.

Boddy (1996) describes a constraint-based distributed scheduling process for air traffic control. Each designated region of airspace is managed by a separate resource manager that allocates spatio-temporal windows to pilots requesting the resource. Musliner and Boddy (1997) describe an application of similar ideas to task distribution for distributed processing. These agents support a similar capability to the modular resource manager we are developing.

Several systems have been developed to support closed-loop plan execution. In contrast with EXEC's current plan execution component, the approach taken in 3T [Bonasso *et al.*, 1997a] has the planner watch over each step of execution. Hence the planner itself serves as an integral participant in the plan execution capability. Bresina *et al.* (1996) describes APA, which has separate components for generation and execution of temporal plans. However, their approach is currently restricted to single resource domains with no concurrency. Reece and Tate (1994) developed an execution agent for the O-Plan [Currie and Tate, 1991] planning system. The combined system supports a plan repair mechanism [Drabble *et al.*, 1996] that is more sophisticated than that supported by EXEC at present, as it allows the planner to edit any unexecuted portion of the currently executing plan. Our redesigned plan execution component will support a similar editing capability, based on the work in O-Plan and also in Cypress [Wilkins *et al.*, 1995]. Finally, Lockheed's Tactical Planning and Execution System (TPES) [Mitchell, 1997] is an interesting related system that supports many execution and replanning capabilities with a high level of human interaction.

5 Summary and Future Work

The Remote Agent Executive demonstrates the capabilities necessary to autonomously control a complex software system. In particular, it supports flexible plan execution, resource management, configuration management, a high-level action-definition language, and system-level fault protection.

In developing EXEC, we have found a number of properties that are desirable for all of the components of an executive. These properties include modular design, adjustable autonomy, robustness, tools for design, analysis, and visualization, and support for testing.

Our current efforts are focused on decomposing EXEC into separate components, as well as enhancing each component. The effort to separate out component capabilities is starting with the resource manager, to which is being added the ability to handle resources as continuous quantities and resource reservations. Other component capabilities will be separated out as their roles and relationships become clarified and as applications require.

Our future efforts involve extending the capabilities of the execution agents in a number of directions to support new kinds of missions, increased reliability, and increased resource utilization. These efforts revolve around support for more flexible representations, distributing execution capabilities, coordination among multiple agents, and improvements in development and verification tools.

Other key directions include increased real-time performance and exploiting the structure of plans to gracefully degrade objectives in the face of failures without aborting the current plans.

6 Acknowledgments

We gratefully acknowledge the contributions of other members of the DS-1 Remote agent team who influenced the design of EXEC: Abdullah Aljabri, Doug Bernard, Ed Gamble, Erann Gat, Ron Keesing, Sandy Krasner, and Reid Simmons.

References

[AAAI, 1997] AAAI. *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, Cambridge, Mass., 1997. AAAI Press.

[Aljabri *et al.*, 1998] A. Aljabri, D. Bernard, D. Dvorak, B. Pell, and T. Starbird. Infusion of autonomy technology into deep space missions: Lessons learned. In IEEE [1998]. To Appear.

[Bernard *et al.*, 1998] D. Bernard, G. A. Dorais, C. Fry, E. B. Gamble Jr., B. Kanefsky, J. Kurien, W. Millar, N. Muscettola, P. P. Nayak, B. Pell, K. Rajan, N. Rouquette, B. Smith, and B. C. Williams. Design of the remote agent experiment for spacecraft autonomy. In IEEE [1998]. To Appear.

[Boddy, 1996] Mark S. Boddy. Contract-based distributed scheduling for a next generation air traffic management system. Technical report, Honeywell Technology Center, 1996.

[Bonasso *et al.*, 1997a] R. P. Bonasso, D. Kortenkamp, D. Miller, and M. Slack. Experiences with an architecture for intelligent, reactive agents. *Journal of Experimental and Theoretical AI*, 9(1), 1997.

[Bonasso *et al.*, 1997b] R. P. Bonasso, D. Kortenkamp, and T. Whitney. Using a robot control architecture to automate space station shuttle operations. In AAAI [1997].

[Bresina *et al.*, 1996] John Bresina, Will Edgington, Keith Swanson, and Mark Drummond. Operational closed-loop observation scheduling and execution. In Pryor [1996].

[Brown *et al.*, 1995] G.M. Brown, D.E. Bernard, and R.D. Rasmussen. Attitude and articulation control for the cassini spacecraft: A fault tolerance overview. In *14th AIAA/IEEE Digital Avionics Systems Conference*, Cambridge, MA, November 1995.

[Currie and Tate, 1991] K. Currie and A. Tate. O-Plan: the open planning architecture. *Art. Int.*, 52(1):49–86, 1991.

[Drabble *et al.*, 1996] Brian Drabble, Austin Tate, and Jeff Dalton. O-plan project evaluation experiments and results. O-Plan Technical Report ARPA-RL/O-Plan/TR/23 Version 1, AIAI, July 1996.

[Firby, 1978] R. James Firby. *Adaptive execution in complex dynamic worlds.* PhD thesis, Yale University, 1978.

[Gat and Pell, 1998] Erann Gat and Barney Pell. Abstract resource management in an unconstrained plan execution system. In IEEE [1998]. To Appear.

[Gat, 1996] Erann Gat. ESL: A language for supporting robust plan execution in embedded autonomous agents. In Pryor [1996].

[Georgeff and Lansky, 1987] Michael P. Georgeff and Amy L. Lansky. Procedural knowledge. Technical Report 411, Artificial Intelligence Center, SRI International, January 1987.

[Hackney *et al.*, 1993] J. Hackney, D.E. Bernard, and R.D. Rasmussen. The cassini spacecraft: Object oriented flight control software. In *1993 Guidance and Control Conference*, Keystone, CO, 1993.

[Havelund *et al.*, 1997] Klaus Havelund, Michael Lowry, and John Penix. Formal analysis of a spacecraft controller using SPIN. Technical report, NASA Ames Research Center, 1997. In preparation.

[IEEE, 1998] IEEE. *Proceedings of the IEEE Aerospace Conference*, Snowmass, CO, 1998. To Appear.

[Levesque *et al.*, 1997] H.J. Levesque, R. Reiter, Y. Lesperance, F. Lin, and R. Scherl. Golog: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31:59–84, 1997.

[Lowry *et al.*, 1997] M. Lowry, K. Havelund, and J. Penix. Verification and validation of AI systems that control deep-space spacecraft. In *Proc. ISMIS'97*, 1997.

[Malin and Leifker, 1991] J. T. Malin and D. B. Leifker. Functional modeling with goal-oriented activities for analysis of effects of failures on functions and operations. *Informatics & Telematics*, 8(4):353–364, 1991.

[Malin, 1997] Jane T. Malin. Statement of interest: Designing model-based autonomous systems for coordinated acquisition and maintenance of models and procedures. In P. Pandurang Nayak and B. C. Williams, editors, *Procs. of the AAAI Fall Symposium on Model-Directed Autonomous Systems.* AAAI Press, 1997.

[McDermott, 1991] D. McDermott. A reactive plan language. Technical report, Computer Science Dept, Yale University, 1991.

[Mitchell, 1997] Steven W. Mitchell. A hybrid architecture for real-time mixed-initiative planning and control. In AAAI [1997], pages 1032–1037.

[Muller and Pischel, 1994] J. Muller and M. Pischel. An architecture for dynamically interacting agents. *International Journal of Intelligent and Cooperative Information Systems (IJICIS)*, 3(1):25–45, 1994.

[Muscettola *et al.*, 1995] Nicola Muscettola, Barney Pell, Othar Hansson, and Sunil Mohan. Automating mission scheduling for space-based observatories. In G.W. Henry and J.A. Eaton, editors, *Robotic Telescopes: Current Capabilities, Present Developments, and Future Prospects for Automated Astronomy*, number 79 in ASP Conf. Series. Astronomical Society of the Pacific, Provo, UT, 1995.

[Muscettola *et al.*, 1998a] Nicola Muscettola, Paul Morris, Barney Pell, and Ben Smith. Issues in temporal reasoning for autonomous control systems. In Wooldridge [1998]. To appear.

[Muscettola *et al.*, 1998b] Nicola Muscettola, P. Pandurang Nayak, Barney Pell, and Brian C. Williams. Remote agent: To boldly go where no AI system has gone before. *Artificial Intelligence*, 103(1/2), August 1998. To Appear.

[Musliner and Boddy, 1997] David Musliner and Mark S. Boddy. Contract-based distributed scheduling for distributed processing. In *Working Notes of the AAAI Workshop on Constraints and Agents*, Providence, RI, July 1997.

[Musliner *et al.*, 1993] David Musliner, Ed Durfee, and Kang Shin. Circa: A cooperative, intelligent, real-time control architecture. *IEEE Transactions on Systems, Man, and Cybernetics*, 23(6), 1993.

[Pell *et al.*, 1997] Barney Pell, Erann Gat, Ron Keesing, Nicola Muscettola, and Ben Smith. Robust periodic planning and execution for autonomous spacecraft. In *Procs. of IJCAI-97*, Los Altos, CA, 1997. IJCAI, Morgan Kaufmann.

[Pell *et al.*, 1998a] Barney Pell, Douglas E. Bernard, Steve A. Chien, Erann Gat, Nicola Muscettola, P. Pandurang Nayak, Michael D. Wagner, and Brian C. Williams. An autonomous spacecraft agent prototype. *Autonomous Robotics*, 5(1), March 1998. To Appear.

[Pell *et al.*, 1998b] Barney Pell, Ed Gamble, Erann Gat, Ron Keesing, Jim Kurien, Bill Millar, P. Pandurang Nayak, Christian Plaunt, and Brian Williams. A hybrid procedural/deductive executive for autonomous spacecraft. In Wooldridge [1998]. To appear.

[Pell *et al.*, 1998c] Barney Pell, Scott Sawyer, Douglas E. Bernard, Nicola Muscettola, and Ben Smith. Mission operations with an autonomous agent. In IEEE [1998]. To Appear.

[Penix *et al.*, 1997] John Penix, Perry Alexander, and Klaus Havelund. Declarative specification of software architectures. In Michael Lowry, editor, *Proceedings of the International Conference on Automated Software Engineering*. IEEE Computer Society Press, November 1997.

[Pryor, 1996] Louise Pryor, editor. *Proceedings of the AAAI Fall Symposium on Plan Execution*. AAAI Press, 1996.

[Reece and Tate, 1994] Glen Reece and Austin Tate. Synthesizing protection monitors from causal structure. In *Procs. AIPS-94*. AAAI Press, 1994.

[Simmons and Whelan, 1997] Reid Simmons and Greg Whelan. Visualization tools for validating software of autonomous spacecraft. In David Atkinson, editor, *Proceedings of the Fourth International Symposium on Artificial Intelligence, Robotics, and Automation for Space (i-SAIRAS)*, Tokyo, Japan, August 1997. Jet Propulsion Laboratory.

[Simmons, 1990] Reid Simmons. An architecture for coordinating planning, sensing, and action. In *Procs. DARPA Workshop on Innovative Approaches to Planning, Scheduling and Control*, pages 292–297, San Mateo, CA, 1990. DARPA, Morgan Kaufmann.

[Wilkins *et al.*, 1995] D. E. Wilkins, K. L. Myers, J. D. Lowrance, and L. P. Wesley. Planning and reacting in uncertain and dynamic environments. *Journal of Experimental and Theoretical AI*, 7(1):197–227, 1995.

[Williams and Nayak, 1996] Brian C. Williams and P. Pandurang Nayak. A model-based approach to reactive self-configuring systems. In *Procs. of AAAI-96*, pages 971–978, Cambridge, Mass., 1996. AAAI, AAAI Press.

[Wooldridge, 1998] M. Wooldridge, editor. *Proceedings of the Second International Conference on Autonomous Agents*. ACM Press, 1998. To appear.